eg: int lock = 0;
in order to lock:

```
entry: while (TS(lock,1)== 1); //spin
exit:  lock = 0; // relies on atomicity of 32-bit assignment => no problem.
```

→ <u>Test-&-Test-&-Set</u> (software method using test&set).
- → Test & set involves a lot of writing to a shared variable (lock). So above method is inefficient.
- → This is a CHEAPER way of doing Test&Set.
- → uses a <u>read</u> spin with a normal read, and only if it is ok, do you use test&set to write.

```
entry: do {
           while(lock == 1); // spin (repetative normal read of a shared variable),
       } while (TS (lock,1 == 1));
```

→ <u>Fetch & Add.</u>
- → older (old Power PC CPUS)
- → not used much anymore

→ <u>Compare & Swap.</u>
- → CAS(x, v_1, v_2) ATOMICALLY sets $x = v_2$ if x is already equal to v_1 and return old value no matter what.

eg: CAS(x, v, x+5);
- → on intel there are for sure 32/64 bit versions. maybe even 8/16-bits.
- → there's a nasty problem under the covers.

SEPT 28, 2010

PAUL HUSSMAN

Last time: "Fair" mutual Exclusion → ticket, bakery.
Hardware primitive to help with mutual exclusion → Test & Set, Test&Test&Set
atomic { → Fetch & Add, Compare & Swap (available on modern intel CPUs).
→ CAS(x,a,b) is x = a? if so set it equal to b. return old value anyway.

<u>The A.B.A. Problem</u> (Problem with Compare & Swap).
IBM engineers were trying to use CAS to build a stack.

```
push(n) {
    do {
        Node t = topOfStack;
        n.next = t;
    } while (CAS (topOfStack, t, n));    // make sure top of stack hasn't changed.
```

$$\begin{tikzcd} n \arrow[r] & \boxed{top} \end{tikzcd}$$

```
pop() {
    do {
        Node t = topOfStack;
        if (t==Null) {
            return null;
        }
        n = t.next;
    } while ( CAS(topOfStack, t, n) != t);
```

- This SHOULD work well, but there's a <u>problem</u>!

example:

stack: ⋮ z ↑ x ← topOfStack

all we check is top of stack

T1	T2
pop	pop
grab x = top	push
n = x.next == z	push

z ↑ y ↑ x

check top. it's still x so we're good. but at this point T2 changed some stuff under x. we lose the value 'y' eventhough CAS worked.

PAUL HUSCMAN

How to fix this problem? (the ABA problem).
- use version #s for data
  → Now CAS with (X, v) where v is the version.
  increment v with each operation
  → the CAS is in the value X version #,
  then we can be sure it's the same X.

Another Primitive
- LL/SC (load-linked, store-conditional) doesn't need version #s.
- on a PowerPC

lwarx (load) → loading with this, loads a value & puts a reservation on the value.
stwcx. (store) → If there are any stores to a reserved value in memory, the reservation is lost.

↳ when stwcx. stores a value.
① checks if reservation is there → fails otherwise.
② returns a condition-code, (reservation is gone after)

- If some one else puts a reservation on before you do stwcx, the reservation gets replaced by whoever loaded it last.

- we have x++ in 2 threads. How do make it atomic?

```
redoit: lwarx r1, 0, x
        addi r1, r1, 1    // r1 ++;
        stwcx. r1, 0, x
        bne redoit:       // redo it if doesn't succeed.
```

With all of these, we have easy-to-build spin-locks
- there is an expressiveness $\boxed{\text{hierarchy}}$
- the idea of "weight-free" synchronization (old concept → Lamport, Herlihy)
  - guarantees a process to complete something in a finite # of steps.
  → consensus # $\boxed{3}$ $\boxed{2}$ $\boxed{1}$ $\boxed{17}$
  want all to agree on a #
  ⇒ weight-free
→ consensus # is the largest # of processes for which a concurrent object (synchronization mechanism) can be used to achieve weight-free synchronization

∞ → CAS, LL/SC
⋮
2 → Test&Set, Fetch&Set
1 → atomic registers
# threads ↑

} theoretical hierarchy.
→ usually probabilities of threads working is good enough but, this is also interesting.

Lamport's ['87] Fast Mutual Exclusion
→ all mutual exclusion algorithms require some shared/communicated data
  → shared data access is expensive (want to reduce it).
  → assumption: Contention is rare (threads don't often fight for the same C.S.) ← (true in practice)
    → Idea: algorithm fast if no contention, but otherwise it can be slow.
→ a thought experiment/logic:
  - S_i = sequence of operations (reads/tests, etc) done by process i
  - in S_i we don't care about local variables, so we only include SHARED variables.
  - every S_i is in fact the same if n (# processes) is arbitrary.
    → as n grows, we must share some code

1st Operation of Si:
→could it be a read?
→all threads/processes can read the same value
⇒doesn't help.

$\underset{=x}{\downarrow}$ $\underset{=x}{\downarrow}$ $\underset{=x}{\downarrow}$

—Therefore it's a WRITE of some variable x.

2nd Operation
→could it be a
{write of x?
read of x?
write of y?
read of y?

→ write (x or y) ⇒ 2 in a row could have been done twice as much in one write
⇒doesn't help.

→ read (x) ⇒ If we write and then read, we may just read what we just write.
→doesn't help.

→observation: all reads must have a write
all writes must have a read (otherwise why write it?)

(skip 3rd)

4th Operation
→ what could be the 4th operation (last operation if we assume just x, y)?
→ if the last operation is a write, doesn't help.
—Therefore the last operation must be a READ.

write x
read y
+(read x
write y)

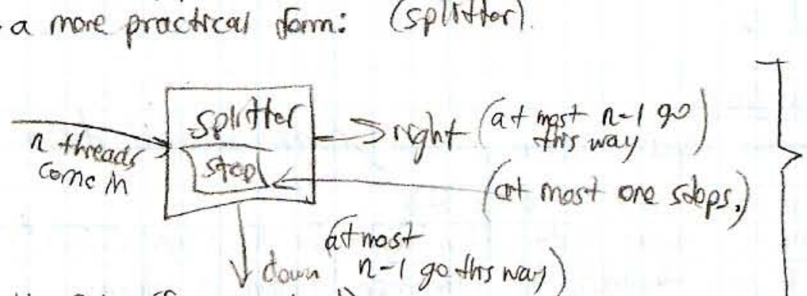
so this is our MIMIMAL solution,
best possible sequence: write x
read y
write y
read x

example code:
line #

```
start: 0   x= id;  //assume ids 1,2,3,... (not 0).
       1   if(y!=0) goto start;
       2   y=id;
       3   if (x!=id) { delay(); // long enough to get another thread through.
       4               if(y!=id) goto start }
```
} entry Protocol.

exit: y=0; //very trivial.

→the delay should be long enough that one thread can make it to the restart parts
→ not very practical.
⇒a more practical form: (splitter)



build a network of splitters to filter threads.

splitter code (found in text).
shared x, y;

```
0   x= id;
1   if(y!=0)  direction = right;
2   else y=id;
3   if (x!=id) direction = down;
4   else direction = stop;
```
} simple code for it.

Sept 28 CONT'D

If one of the threads is stopped
ie has made it through,
→ and another shows up,
→ the rest go off to the right.
If two threads show up at the splitter at the same time.
→ only one will end up with x as the id, so at least one goes down.

use: moodle.cs.mcgill.ca/moodle instead of webct

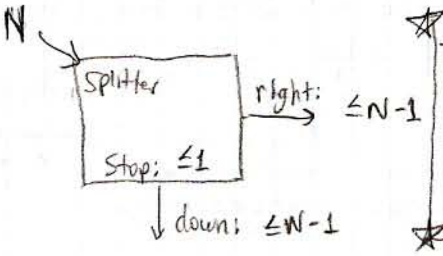
COMP 409
SEPT 30 2010

[PAUL NUSSMAN]

Last Time: Ways of making locks for Mutual Exclusion → Lamport's Fair Mutual Exclusion → cheapest version?
→"theoretical arguement", implementation with delay kind of sucks, so we use the splitter idea!

Splitter Code:

```
x = id;
if (y != 0) goto right
else y = id
     if (x != id) goto down
     else stop
```



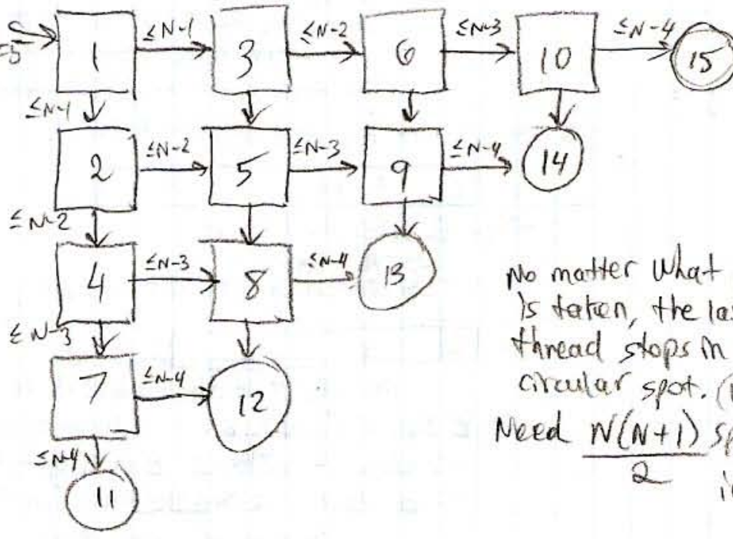
→ we can prove that at most 1 can be stopped.
→ EXERCISE → use contradiction. assume 2 threads have stopped. should be ≤ 1 page proof.

what can we do with the splitter?
→ can create networks of splitters
→ can solve a problem → thread renaming
→ usually we think of these (thread id's) as: { 1....N / 1....∞ } eg: you might have T1, T2, T240672 active.
→ the range of thread ids is not necessarily static/bounded. (current, valid ids)
→ with splitters, we can create a Thread Renaming Network.

Thread Renaming Network:
eg: N=5 → only 5 threads are being used at a time.
→ idea: each thread that stops gets the id of the splitter in which it stopped.

→ Splitters can be treated as objects, so you don't even really need to know the code for it, just how it works, and you can then build things with them.

→ This doesn't mean we should use splitters everywhere and all the time, but it's an interesting idea. → can be useful.

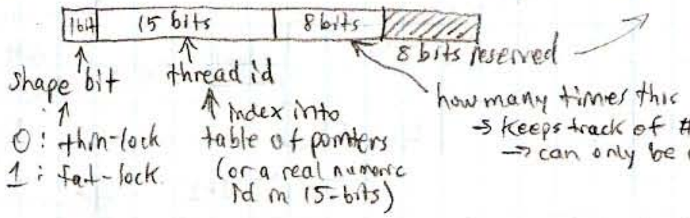


No matter what path is taken, the last thread stops in a circular spot. (11-15).
Need $\frac{N(N+1)}{2}$ splitters, ie $O(n^2)$

LOCK DESIGN.
→ how are locks actually implemented?
In Java
→ spin-locks → expensive if you spin for a long time. → good for short waits.
→ blocking synchronization instead of spinning, we let the thread sleep, relying on a wake-up call.
→ sleeping is lots of work (gotta deal with OS) good for long waits
Ideally we should adapt → choose which one to use in each case.
→ Problem: we don't always know how things will shape up → depends on scheduling etc.
→ We need a lock that will actually switch modes.
1998 → Bacon, Kamura, Morthy, Serano. (when Java was still young)
→ wanted a cheaper, faster lock than mutexes
→ they analyzed programs → how often are some threads doing certain behaviors?
→ they came up with a hierarchy:

→ heirarchy: most frequent: → a single thread locking & unlocking (by far most frequent).
→ shallow recursive locking. (multiple locks at once)
→ deep recursive locking.
→ shallow contention → other threads want in, but we hold the lock,
least frequent: (by far) → deep contention

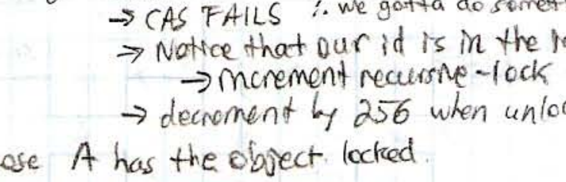
→ the focused on the top two scenarios. the other ones are just handled in a default way.
→ we assume a heavy-weight mechanism exists for locking → pthreads mutexes
→ we will need CAS (compare & swap). → add a lock word in the header of each object,
→ every object must have the ability to be locked.
→ the "lock-word" stole 24-bits from something else that only needed 8 bits. ∴ objects didn't need to grow



1bit
15 bits
8 bits
8 bits reserved
shape bit
thread id
0: thin-lock
1: fat-lock
index into table of pointers (or a real numeric id in 15-bits)
how many times this lock is locked?
→ keeps track of # of shallow locks.
→ can only be done 256 times.

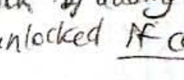
[Note: this was a 32-bit machine]

→ no thread has id of 0, we can initialize unlocked objects to: [0| 0 | 0 |▨]
→ to LOCK this,
test if id is 0
if so set it to our id. } MUST BE ATOMIC ⇒ CAS(lock, 0, id)
→ if it succeeds, we own it. ⇒ [0| id | 0 |▨]
→ to unlock it, just set id back to 0.
→ to lock recursively,
– 1st time: [0| id | 0 |▨]
– 2nd time:
→ CAS FAILS ∴ we gotta do something else
→ Notice that our id is in the id field
→ increment recursive-lock by adding 256 to whole thing to get past the end.
→ decrement by 256 when unlocked if count > 0.
→ suppose A has the object locked.



→ if B tries to lock it,
→ CAS Fails.
→ TRANSITION INTO A FAT LOCK!

index to table of pointers to mutexes (pthreads).
→ B spins (relying on the hope that A leaves soon).
→ when A unlocks, B will get in.
→ B does the transition to a fat lock (if necessary). (it double checks in case its already been done).
→ allocate a mutex
→ put it inside
→ change the shape bit
→ If it was already a fat-lock, then just lock the mutex.
→ thereafter, its just a regular mutex.
→ basic scheme for thin-locks! thin lock → fat lock

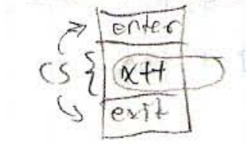


→ There are some improvements:
Onodera, Kawachiga: Found a way (in 1999) to allow for this to happen.
– the transition from fat to thin was figured out with the addition of a couple bits.
→ improved the spinning.
→ this is what happens for real in the JVM. but deflation is not necessarily immediate.

COMP 409
SEPT 30, CONT'D

PAUL HUSSMAN

Race Conditions



→ every shared variable access must be "protected" by synchronization of some form.
→ synchronization order
→ intra-thread order:

x=y; ① ↓ static, from program.
x=z+1; ②

→ synchronization order;
→ order between synchronization actions.
→ order in which things lock & unlock for each other. (dynamic).

OCT 5, 2010

PAUL HUSSMAN

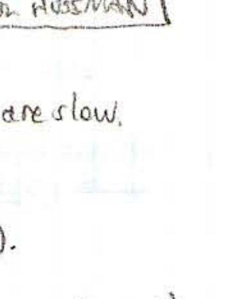
Last Time: lock design (Java) → Bacon's Thin locks.
→ Java locking is efficient. → common path is quick, uncommon paths are slow.

Race Condition

→ correctness: Concurrent programs should NOT have race conditions!
→ Race Conditions are (usually) considered errors! (but not always).

→ Definition of Race Condition.
- Program order of execution (line-by-line) → Intra-thread order (obvious, trivial).
- Synchronization Order (this is a dynamic order → can't tell by looking at code!
→ how are sync operations actually ordered at run-time?

depends! $\begin{bmatrix}T1\\T2\\T3\end{bmatrix} \begin{bmatrix}T1\\T3\\T2\end{bmatrix} \begin{bmatrix}T3\\T2\\T1\end{bmatrix}$ ... etc.



- A race condition is:
A data race in a multithreaded program occurs when 2 threads access the same data from the same memory location with no ordering constraints, such that at least one of these accesses is a WRITE.

- To prevent race conditions:
In practice, all shared data should be protected by synchronization.

```
synchronized ((★)) {        synchronized ( (★) ) {      (for the same object)
   =x                           = x
}                            }
```

- How do we do it without synchronizing?
→ the VOLATILE keyword also protects shared data!
It auto-synchronizes every operation on x.

```
lock
  = x
unlock
```

→ Note: this has no impact on atomicity
atomicity is a different issue.

★ so if you have volatile int x;
and if you have x++; there's no race condition, but it might not give the right answer due to lack of atomicity.

→ Note: if you use synchronized to protect x everywhere, you shouldn't declare x as volatile because its redundant.

So: if 2 threads try to do x++ at the same time, the only way to ensure it works properly is to synchronize the x++ procedure, even if x is declared volatile.

Blocking Mutual Exclusion

→ spinning is a mutual-exclusion mechanism ⇒ cheap if spin is short, but otherwise expensive.
→ it is better if the thread goes to sleep instead of spinning. } VERY EXPENSIVE, but could be worth it.
BUT we want someone to wake us up once the lock is free.
→ so spinning is good for short waits, and sleeping is good for long waits.
→ Problem:

```
        T1                                   T2
     (locked) {   > not atomic          if anyone is sleeping?
        sleep;                            ↳ no.  (happens here:)*
  enter } else lock();
                                       → NOW NO-ONE WAKES ANYONE UP!
      [CS]                               ⇒ PROBLEM.
                                           → lost wake-up problem due to
  exit  → check if anyone is sleeping        failure of atomicity!
          if so wake them up.   > not atomic
```

→ How to deal with the LOST-WAKE-UP POTENTIAL
→ SEMAPHORE → ⟨Dijtestra 1960's⟩
→ prevents lost-wakeups.
→ basic semaphore:
→ value (integer)
→ 2 operations (Down and Up) → (DOWN) P( ), (UP) V( )

→ Down (P)

```
   → check the value;
     if (value is 0 or less) {
        wait until it's not;        } ATOMIC
     }
     once it's above zero,
     decrement the value.
```

- In other words;

< await (value > 0); value --; >

→ Up (V)
→ increment the value; and that's it
→ so we do DOWN operations until 0, and any further we wait.
→ when we do UP operations we may allow a waiting thread to proceed.
→ Something to keep in mind
→ We don't have such great control as it may seem.

```
  eg:   s.down() ← T2
                 ← T3
          | | ⇒ T1 reduce to 0
          |
          T1
        s.up() lets some threads in. WHICH ONE? → undetermined.
```

→ who gets in next is NOT determinitstic from the pool of waiting threads.
→ BARGING IS POSSIBLE! ie budding in line. → some threads could wait a long time!
⇒ some other non-waiting thread could get in ahead of sleeping threads.
- THEREFORE NO FAIRNESS IS GUARANTEED!

→ A regular semaphore is considered a COUNTING SEMAPHORE (which is what we just discussed)
→ assumes values are above 0.

→ BINARY SEMAPHORE
→ value can only be 0 or 1.
→ Just like a mutex or a synchronized block.
⇒ initialize as 1, decrement to enter and increment to exit.
→ they are actually different from mutexes (just conceptually similar)
→ a mutex has OWNERSHIP, but that's not built into semaphores.

COMP 409
OCT 5 CONT'D

PAUL HUSMAN

→ mutexes & synchronized blocks are OWNED
  so whoever locks something is the same one to unlock it.
  → semaphores don't guarantee that at all.

→ SIGNALLING SEMAPHORE.
  → special case of binary semaphore.
  → starts at 0
    → first intended operation is a DOWN operation, so that thread waits,
    → someone else comes along and does an UP operation, signalling the other thread to wake up.
    → not necessarily used for Mutual Exclusion.
    → this is just used to say "ok, you can go now".

→ Now we can build things
  eg: BARRIER SYNCHRONIZATION

```
    S1 = S2 = 0;
    T1              T2
    S2.up()         S1.up()
    S1.down()       S2.down()
```

if T1 shows up first, it waits for T2. ⎫ Then they
if T2 shows up first, it waits for T1. ⎭ both go

→ Split binary Semaphore.
  → set of binary semaphores
  → only one semaphore (at most) has value 1 at any time. The rest are all 0's.
  → we need to implement this in most cases ourselves   00010000
  - What are these good for?   or 00000000
    → Easy to implement a solution style that's called "pass the baton".
      → you can only run if you have the baton.
      → the thread that saw the 1 gets to execute
        down() down() down() down() up (on another)
      → when the thread finishes it does an up on another semaphore and lets
        the thread associated with that other semaphore to complete down()
        ie it passes control along to another specific thread → pipe-line style.

OCT 7 2010

Last time: locking implementation/design/issues
  Race conditions: can prevent it by making everything volatile! → not always a good idea though, less efficient for compiler.
  Note: in Java you can't declare anything as volatile. eg: can't declare an array as volatile. It only declares the pointer to the array as volatile!

→ Continuing with Semaphores.
  → Recall Semaphores are unfair → order of threads is not deterministic
    → some are fair, but it comes with a price.

SEMAPHORE EXAMPLES

→ Use a semaphore as a lock.
  classic example: Producer - Consumer. with shared data. eg: type letters, P stores them, C prints them.
    Rules: we should consume only produced items, not from empty buffer.
      never consume an item twice
      producer: must not over-produce and overfill the buffer.
      should consumption be in the same as production order? → keyboard yes, sometimes no.

```
  semaphore lock = 1;
  int buffer;
  boolean filled = false; // is anything in buffer?

   P()                                 C()
   while (true) { lock.down()          while (true) {  lock.down()
     if (!filled) {                      if (filled) {
         buffer = produce();                 consume(buffer);
     }   filled = true;                  }   filled = false;
   lock.up()                             lock.up()
   }                                   }
```

THIS CAN BE MADE MORE EFFICIENT ⟋

→ making Producer-Consumer more efficient
⇒ this version uses two counting semaphores:

```
semaphore spaces = N; // for a buffer size of N.
semaphore filled = 0; // start off empty.
int inserted = 0; // place where there is a space
int removed = 0;
int buffer [N];
P( ) {      // Producer
    while(true) {
        Down (spaces);
        buffer[inserted] = produce ( );
        inserted =(inserted +1) % N;
        Up (filled);
    }
}
```

⇒ Producer can produce only if there's space
→ Consumer consumes while still filled.

```
C( ) { // Consumer.
    while (true) {
        Down (filled);
        consume (buffer[removed]);
        removed =(removed+1)% N;
        Up (spaces);
    }
}
```

→ Monitors & Condition Variables

→ Using semaphores tends to be tricky (easy to mess up by forgetting an up or a down)
→ Semaphores wrap together 2 ideas: MUTUAL EXCLUSION and SIGNALLING (communication telling what thread to go when)
→ Monitors are a solution to these problems.

→ Monitors seperate signalling and Mutual Exclusion
→ Monitor is actually an ABSTRACT DATA TYPE (ADT).
→ We have PRIVATE DATA, and FUNCTIONS that operate on that data, which are the only things that touch the private data. These functions must be mutually exclusive.
Note: Java does NOT have MONITORS(TM) It has synchronized from which monitors can be constructed. It's up to the programmer to make sure the data is actually private with synchronized methods.
→ so far we have mutual exclusion
→ how do we communicate between threads (signalling)?
→ we use a CONDITION VARIABLE.
→ condition variables are always associated with monitors.
→ when we're in a monitor, we want to check the condition, and pause at certain points without continuing on until someone else says it's ok.
→ come across condition
→ give up the monitor (but keep current place)] Wait.
→ let other threads do something
→ let us know when the condition (might be) true] signal
→ example of idea:

```
T0                              T1
enter monitor (lock)
  if (!b) {
     unlock()
     sleep()  ←――――――――→  lock()
     lock()                  b= ....
                             notify()
   ―――――― - - - - - - - -  unlock()
  (continue)
```

problem: Possible loss of wake-up!
↓ solution

→ Condition variables provide an ATOMIC way of doing unlock() & sleep().
THIS IS HOW IT WORKS.

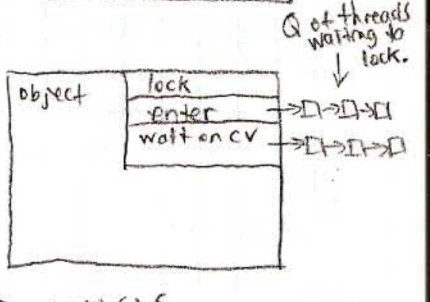
```
T0
lock, check condition
  if not true
    wait() // atomically unlocks
              and sleep
      lock()
unlock
```

```
T1
- lock
- change something.
- signal (NOTIFY) // no longer sleeping
                     but not in monitor.
- unlock // some thread can now get in. dont know who.
```

COMP 409
OCT 7 CONT'D

PAUL HUSSMAN.

- How COULD this be implemented?
  - → every Java object has a lock.
  - → every Java object has a single, unnamed CONDITION VARIABLE.
    - → on this condition variable we can call wait() and notify().
      - → wait() unlocks (this), goes to sleep, then locks (this) where the unlock & sleep are done ATOMICALLY.
    - → notify() → wake up one thread that did wait() on <u>this</u>.



COMP 409
OCT 12, 2010

<u>Last Time</u> → finished up semaphores
→ wrapped up signalling & mutual exclusion.
→ cleaner/better idea: Monitors. → encapsulation → A.D.T.
→ Monitors separate signaling & mutual exclusion.
   monitor methods are mutually exclusive. eg: via a mutex.
→ condition variables → built into Java objects.
   → mechanism:

```
T0                                              T1
{                                               }
locks monitor                                   ↓ block
   if (   ) //check state
          wait(); //atomic unlock + sleep,      ↓
                  then re-lock.
unlock                                          notify() //wakes up 1 waiting thread if any.
                     lets this continue         }
                                                unlock.
```

```
wait() {
   - puts into wait Q then
     will unlock.
}
notify {   ← dont know which,
   gets ONE thread from the
   wait Q and puts into
   the enter Q
}
```

eg: <u>Producer / Consumer with monitor.</u>

```
class n {
   private boolean filled; // does buffer have data in it?
   private int    buffer;

   synchronized void produce() { //synchronize to ensure mutual exclusion.
      if (filled) //typically, this should be [while(filled);] *
         wait();
      buffer = ....; // put something into buffer
      filled = true; // flag as filled.
      notify(); // wake up a waiting consumer
   }
   synchronized void consume() {
      if (!filled)   // should be [while(!filled);] *
         wait();
      consume(buffer);
      filled = false;
      notify();
   }
```

* wait(); is subject to "spurious wake-ups" (a thread spontaneously wakes up)
so it is ALWAYS necessary to double-check the condition.

```
if (b) {                           while(b) {
   wait(); ✗ (WRONG!)     ⇒           wait(); ✓ * (if it wakes up spontaneously
}                                  }               it will go back to sleep.)
```

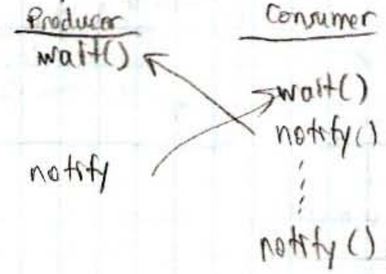
So spurious wake-ups are a concern.
→ it is expensive to ensure NO spurious wake-ups (lots of little "corner cases")
→ for easy initial implementation

TRIVIAL Implementation of Condition Variables.

```
wait(){ unlock();
        sleep(randomTime());
        re-lock();
}
notify(){
        no-op.;
}
```

→ Broadcast signals.

eg: Multiple Producers & Consumers.



⇒ with multiple consumers how can we be sure to wake a producer?
we need a way to reach the intended thread.
⇒ BROADCAST NOTIFICATION.

notify() // how do you know if it wakes up a producer or consumer?

Solution: wake up <u>all</u> threads
Java: notifyAll(); ⇒ wakes up everyone waiting on this condition variable
→ the ones that should not have been woken up wake up, otherwise will sleep again.

Monitor Semantics
→ In Java, you cannot notify unless you hold the lock. (must be in a synchronized block).
→ In pThreads, mutexlock(&m); cond-wait(&m, &cv);
mutexunlock(&m); cond-signal(&cv); // you can signal when unlocked

⇒ we use in Java & Pthreads something called
★ SIGNAL-AND-CONTINUE → (MOST COMMON) aka MESA MONITORS.
→ notifier wakes up someone else & it can continue inside the monitor

SIGNAL-AND-WAIT.
→ notifier is kicked out of the monitor and the signallee enters the monitor w/out competing for the lock.
→ notifier needs to compete for the lock to continue on.

SIGNAL-AND-URGENT-WAIT.
→ like signal & wait, but when the signallee leaves the monitor, the notifier gets back in. ie the notifier gets put in the "front" of the queue of threads contending for the monitor.

SIGNAL-AND-RETURN / SIGNAL-AND-EXIT
→ notifier thread exits the monitor at the same time it signals.

```
synchronized ___() {
   :
   notify();   ] signal & exit together.
}
```

AUTOMATIC SIGNALLING
→ wake up everyone every time there's a monitor-exit. (implicitally)
→ a bit more expensive, but easy.

IMPLEMENTING SEMAPHORES WITH MONITORS // before Java had semaphores you had to make your own.

```
public class Semaphore {
    private int count;
    public Semaphore(int init) {
        count = init;
    }
```

COMP 409
OCT 12 CONT'D

PAUL [illegible]

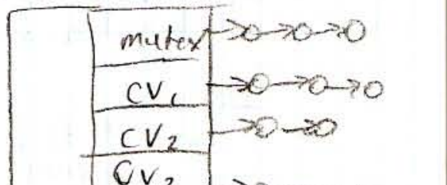
```
public synchronized void down() {
    while (count == 0) {
        wait();
    }
    count --;
}
public synchronized void up() {
    count ++;
    notify();
}
```

In PThreads we can have more than one Condition Variable associated with the <u>same monitor</u>!

<u>Multiple Condition variables in Producer/Consumer</u>

```
mutex m;
buff [N];
int front, rear, count;
CV notFull;
CV notEmpty;
Producer() {
    mutex-lock(&m);
    while (count==N) {
        cond-wait(&m, &notFull);
    buff[rear] = ......;
    rear = (rear+1)% N;
    count ++;
    cond-signal(&notEmpty);
    mutex-unlock(&m);
}
```

```
Consumer() {
  mutex-lock(&m);
  while(count==N){
    cond_wait(&m, &notEmpty));
  consume(buff[front]);
  front = (front+1)%N;
  count --;
  cond-signal(&notFull);
  mutex-unlock(&m);
}
```



Java only has one CV but pthreads can have many.

<u>We can implement the same thing in Java!</u>

```
public class CV{
    public void CV_wait (mutex m) { // mutex can be implemented as a binary semaphore.
        try {
            synchronized(this) {
                m.unlock(); } atomic w.r.t. the thread that could cause a problem.
                wait();       effectively atomic. no one can notify at this point.
        } catch (InterruptedException ie) { } // ignore exception -> treat as spurious wake-up.
        finally { // lock no matter what
            m.lock(); //executed no matter what.
        }
    }
    public synchronized CV_notify() {
        notify();
    }
```

-OR- (same thing)

```
public CV_notify() {
    synchronized(this) {
        notify();
    }
}
```

OCT 14, 2010

Readers and Writers

→ given a database, there are different processes going on → something to read, some trying to write.
→ don't want 2 threads writing at the same time (in case they write the same thing).
→ don't want something reading while something is being written
→ multiple reads with no writes should be fine.

Non-Optimal solution:
→ make a lock so only one thing can R/W at once. → does the right thing.
→ Problem: doesn't allow for multiple readers.
→ in databases, multiple reading is essential!

→ write actions should wait until current reads are done, prevent reads and then happen one at a time (serialized), and then let reads back in.
→ ASSYMETRIC Mutual Exclusion.

→ Semaphore-based solution: [Weak-Reader-Preference]

```
int readers; // count of readers.
BinarySemaphore r = 1, rw = 1;
```

Reader

```
while(true) {
    r.down()
    reader ++;
    if (readers == 1) {
        rw.down();
    }
    r.up();
    read(); // whatever readers do.
    r.down();
    readers --;
    if (readers == 0) {
        rw.up();
    }
    r.up()
}
```

Writer

```
while(true) {
    rw.down(); //unique mutually exclusive access to the database.
    write();
    rw.up();
}
```

→ One concern
→ If there's an infinite stream of readers, writers "starve" (wait forever).
→ this is known as the (weak) Reader's Preference.
→ it is weak because if 2 come at the same time (one R, one W), there's no guarantee which one gets in first.

→ Monitor-based Solution → writers never starve, but new readers can starve. [WEAK WRITERS PREFERENCE] (more acceptable)
→ need some data to keep track of readers & writers
nr → number of readers (0 or more)
nw → number of writers (0 or 1).
ww → number of WAITING writers (0 to more).

```
read() {
    synchronized(this) { //mutex lock
        while (nw != 0 || ww > 0) {
            try { wait(); } catch (_) {}
        }
        nr++;
    }
    readSomething();
    synchronized(this) {
        nr--;
        notifyAll();
    }
}
```

```
write() {
    synchronized(this) {
        ww++;
        while (nr>0 || nw>0) {
            try { wait(); } catch (_) {}.
        }
        ww--;
        nw ++;
    }
    writeSomething();
    synchronized(this) {
        nw--;
        notifyAll();
    }
}
```

COMP 409
OCT 14 CONT'D

PAUL HUSSMAN

Fair Solution (sort of ideal most of the time...)
- → neither R or W can starve.
- → 2 Condition Variables ⇒ nb: Ignores spurious wakeups! → we must figure this out.

```
int nr
int nw
int ww
int wr
mutex e
CV okread
CV okwrite
```

Reader

```
lock e;
if(nw > 0 || wr > 0) {      // fix this for spurious
    wr++;                       wakeups
    wait (e, okRead);        ↳ use while loop
}else {                        But there's an
    nr++;                      else. Figure it out!
}
unlock e;
readSomething();
lock e;
nr--;
if(wr > 0 && nr == 0) {
    ww--;
    nw++;  // nw = 1;
    signal(okWrite);
}
unlock e;
```

Writer

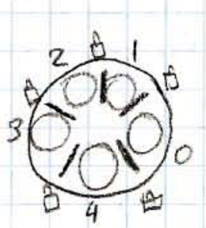
```
lock e;
if ( nw > 0 || nr > 0 || wr > 0) {
    ww++;                         // fix this
    wait (e, okWrite);            for spurious
} else {                          wakeups.
    nw++;
}
unlock e;
writeSomething();
lock e;
nw--; // nw = 0;
if(wr > 0) {
    nr = wr;
    wr = 0;
    signalAll (okRead);
} else if (wr > 0) {
    ww--;
    nw++;
    signal (okwrite);
}
unlock e;
```

→ readers & writers are not only useful for databases.

DEADLOCK & TERMINATION.

- Dining philosophers. 5 of them. → thinking, hungry, eat
  - table has 5 plates, one chopstick between each plate and they each need 2 chopsticks to eat.
  - How can we synchronize their actions to avoid problems?

Solution 1. - A single lock for the whole table.



```
- Pi : think
       Down (globalLock)
     → eat ←
       Up (globalLock)
```

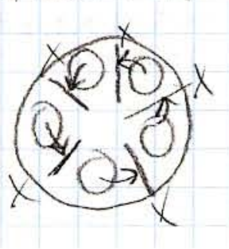
- drawback → only one philosopher eating at a time. (Not a great solution).

→ what resources do we care about? → chopsticks!

Solution 2: One lock per chopstick.

```
→ Pi think
     Down (ci)
     Down (c(i+1)%5)
     → eat ←
     Up (c(i+1)%5)
     Up (ci)
→ 2 can eat at once, but problem.
```

→ they all want to eat at once, and follow steps...



DEADLOCK!
philosophers starve

Solution 3 - global lock plus the chopstick locks.

```
Pi  think
    Down (global)
    Down ($c_i$)
    Down ($c_{(i+1)\%5}$)
    Up(global)
    → eat ←
    ~~Down (global)~~
    Up ($c_i$)
    Up ($c_{(i+1)\%5}$)
    ~~Up(global)~~
    dumb idea!
```

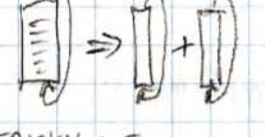
→ this doesn't work in the following scenario:

```
P1  Down (global)
    Down (C1)
    Down (C2)
    Up (global)

P2  Down (global)
    Down (C2) // stuck here
              // global is stuck too!
    // P1 eats forever!
```

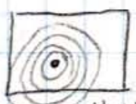
$\boxed{\text{COMP 409, OCT 19, 2010}}$ - Assignment 2 is out now. Grades will be up soon for A1. $\boxed{\text{PAUL HUSSMAN}}$

→ Q1 → monitor semantics
→ Q2 → use semaphores to implement a loop with 2 threads.



In the body, it sometimes refers to a previous iteration, so it must be controlled.

→ Q3 ⇒ TRICKY ONE.
OBJECTS IN (n+1)-D space.
eg: 2D space + time. (move arbitrarily in 3D space, forward in time).
modelled on a donut ⇒ closed space
so it wraps around ⟶



How far can a dot move? one radius length per time.

- objects exist at different time spans.
  → must maintain a consistant universe.
    → shouldn't see things from the past or future
      - If they're all at the same time, no worries
      - if one point's radius can reach another, then the other has to wait and see if it actually goes there
      - otherwise it can move
  → you CAN have two points in the same place at the same time
  → make sure you don't see anything that didn't happen
  → use monitor-based synchronization,
  → If one point has to wait, it should wait on a condition variable
  Hardest part → distance between 2 objects in a taurus.

Back to Deadlocks
- Dining Philosophers. → Recall Solution 3 fails sometimes too.
  - we then try removing the global lock for releasing a chopstick. Works, but only one eats at a time.

```
eg: P1  Down (global)
        Down (C1)
        Down (C2)
        Up (global)
    P2  Down (global)
        Down (C2)  → stuck while P1 eats
        ∴ everyone's stuck while P1 eats
          since P1 has global lock.
```

COMP 4109
OCT 19 CONT'D

PAUL HUSSMAN.

- Solution 4 → grabbing chopsticks in random order.
  → RANDOMLY choose L or R chopstick first.
  → This works with high probability. (for a while)
    → but eventually a similar deadlock occurs
- Solution 5
  → Always leave 1 empty seat.
  → initialize a new semaphore to 4, and down(seat) to sit down.
  → So:

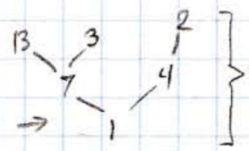
```
think
Down (seats)
Down (ci)
Down ((i+1 %5)
→eat←
Up (ci)
Up (Ci+1%5)
Up (seats)
```

Solution 6
  → ACQUIRE & RELEASE.
    → Philosophers give up chopsticks if they can't get both.
    → Then wait & repeat.
    → Eventually works with high probability.
    → has unfortunate random delay ⇒ the "wait"

Solution 7
  → ORDER THE RESOURCES.
  → If we give all resources an ordering (a partial ordering) and we always acquire them in order, WE CANNOT HAVE DEADLOCK.

$$\begin{array}{ccccccccc} P_1 & & P_2 & & P_3 & & P_4 & & P_5 \\ c_1 & < & c_2 & < & c_3 & < & c_4 & < & c_5 \cancel{<} c_1 \\ & & & & & & & & c_1 < c_5 \end{array}$$

- In general, make up an ordering of your locks →



} nobody can wait for itself

COFFMAN'S CONDITIONS - 1971
- Model: Resource Deadlock. (deadlock that occurs as things fight for resources).
  All of these must be true for deadlocks to be possible:
  1) Serializably Reusable resources → multiple threads acquire & release
  2) Incremental Acquisition → acquire multiple resources incrementally without releasing in between
     ie: grab first chopstick and don't release until after you get the second.
  3) No preemption → once you have a resources, its yours until you choose to release it.
  4) Dependency cycle → circular chain of processes, each holding a resource that the next one wants. ("wait-for graph" is circular)

- To avoid deadlock, avoid at least one of Coffman's conditions
  → of course, some conditions are harder to change than others.
    1) need chopsticks → otherwise problem fundamentally changes → Hard to change this
    2) need 2 at a time to eat. → otherwise problem fundamentally changes → Hard to change this.
    3) steal chopsticks from neighbor (can sometimes be changed)
    4) Easiest thing to change.

- OTHER kinds of deadlock (other than resource deadlocks).

→ Communication Deadlock

- two trains waiting to cross but they each wait till its clear, but its never clear because they're both there.

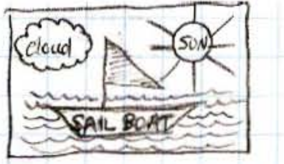
→ divergence

eg: while(true); //infinite computation.

→ looks stuck, but it's not waiting for anything/anyone.

→ FLAW in a sequential program

⇒ could be valid in a concurrent program → but don't use this! wastes CPU for no reason.



THREAD TERMINATION.

→ Safety issues exist with ASYNCHRONOUS termination.

$$\begin{tikzcd} T_1 \arrow[r, bend left, "stop"] & T_2 \; m.lock() \end{tikzcd}$$

→ we don't really know the state of T2

→ what locks it holds?

→ what function/private state it's in?

eg: In early versions of Java, 64-bit integer operations were done by function calls.

→ In general, it's too dangerous since you can't be too sure exactly what's going on.

→ instead set a flag,

eg: pThreads

→ POSIX lets you call stop which sets a flag. SYSCALLS check the flag.

→ this is called CANCELLATION.

→ when you cancel a thread it sets the flag

→ flag is checked frequently, How frequently? ⇒ Nearly every BLOCKING syscall.

- Error Interrupt Code EINTR ⇒ every syscall that could return this checks the flag.
- Doesn't check on mutex-locks. (purely for performance) but there's an API to check yourself.